Reconfigurable video coding: a stream programming approach to the specification of new video coding standards

Jörn W. Janneck, Marco Mattavelli, Mickael Raulet, Matthieu Wipliez

Reconfigurable Video Coding — a Stream Programming Approach to the Specification of New Video Coding Standards

Jörn W. Janneck
Department of Automatic Control
Lund University
SE-221 00 Lund, Sweden
jwj@acm.org

Marco Mattavelli
Microelectronic Systems Lab
EPFL
CH-1015 Lausanne
marco.mattavelli@epfl.ch

Mickael Raulet
IETR/INSA Rennes
F-35708, Rennes
France
mraulet@insa-rennes.fr

Matthieu Wipliez
IETR/INSA Rennes
F-35708, Rennes
France
mwipliez@insa-rennes.fr

ABSTRACT

Current video coding standards, and their reference implementations, are architected as large monolithic and sequential algorithms, in spite of the considerable overlap of functionality between standards, and the fact that they are frequently implemented on highly parallel computing platforms. The former leads to unnecessary complexity in the standardization process, while the latter implies that implementations have to be rebuilt from the ground up to reflect the parallel nature of the target.

The upcoming Reconfigurable Video Coding (RVC) standard currently developed at MPEG attempts to address these issues by building a framework that supports the construction of video standards as libraries of coding tools. These libraries can be incrementally updated and extended, and the tools in them can be aggregated to form complete codecs using a streaming (or dataflow) programming model, which preserves the inherent parallelism of the coding algorithm. This paper presents the RVC framework and its underlying dataflow programming model, along with the tool support and initial results.

Categories and Subject Descriptors

D.3.0 [**General**]: SubjectsStandards; B.5.2 [**Design Aids**]: Automatic Synthesis

General Terms

Standardization

MMSys'10, February 22–23, 2010, Phoenix, Arizona, USA.

Keywords

Reconfigurable Video Coding, Dataflow programming, Motion Picture Expert Group

1. INTRODUCTION

The standardization efforts in the video coding field, besides their main objective of guaranteeing interoperability of compression systems, have also aimed at providing appropriate forms of specifications for wide and easy deployment. ISO/IEC SC29/WG11, better known as MPEG, is a working group in ISO/IEC that has produced many important, innovative and successful standards following these objectives.

While at the beginning MPEG-1 and MPEG-2 were only specified by textual descriptions, with the increasing complexity of algorithms, starting with the MPEG-4 set of standards, C or C++ specifications, also called *reference software*, have become the formal specification of the standards. However, descriptions composed of non-optimized non-modular software packages have begun to prove limiting. Since in practice they are frequently the starting point of an implementation, system designers must rewrite these software packages not only to try to optimize performance, but also to transform these descriptions into an appropriate form adapted to the current system design. Existing monolithic specifications hide the inherent data flow structure of the video coding algorithms, which is a significant impediment for efficient implementation on any class of target platform. Furthermore, the intrinsic concurrency and parallelism of any modern video algorithm, which is an essential characterictic for the efficient implementation on the next generation of multicore platforms, is completely obscured by reference SW written using pure sequential languages. The required analysis and rewriting of the specification is difficult and time consuming, as it involves parallelizing a highly complex sequential algorithm relying heavily on global storage into a parallel implementation that exploits the manifold forms of concurrency in it, and whose observable behavior is identical.

At the same time, the evolution of video coding technologies leads to solutions that are increasingly complex to design and present significant overlap between successive standards. Traditionally, the monolithic nature of the reference software required complete rewrites of the code for new standards and failed to take advantage of the overlap in functionality, effectively obscuring what is new and what is common between an old and new standard specification.

Another problem of the current specification form, and a consequence of the wide variety of video coding algorithms, is the selection of the subsets of coding algorithms used by a specific application domain. These (sub-)sets are also known as "profiles" in MPEG. The "a priori" specification of a small number of such profiles has become very problematic. Since some algorithmic components are rather complex and only to be implemented on certain devices, or are not required for all applications, they are included only into standardized "profiles", which simply constitute subsets of coding algorithms providing codecs implementations satisfying specific application constraints. Interoperability is thus guaranteed at the level of these standard profiles. However, such "trade-off" subsets prevent the possibility of optimally satisfying a variety of specific applications, whereas the specification of too many profiles would result in an obstacle for guaranteeing interoperability. In any case the lack of modularity and encapsulation properties of the specification form does not help to support more flexible and dynamic ways to select such algorithm subsets to satisfy other specific application requirements.

Those requirements could include anything besides coding efficiency, which is the traditional focus of current standardization efforts. Although coding efficiency is fundamental, other trade-offs, such as coding performance (in terms of overall computational effort or latency), are not efficiently realizable staying within current standard profiles. Moreover, the possibility of seeking these new trade-offs for codecs and the motivations that lie behind such search of new "profiles" are increased on the one hand by the large number of coding algorithms available in existing standard technology (MPEG-1, MPEG-2, MPEG-4 Part 2 and part 10 including AVC and SVC) and on the other hand by the increasing number of application domains employing video compression, in addition to digital video broadcasting and storage, which are looking for very specific optimizations.

Last, but not least, the current scheme for the definition and standardization of new video coding technology results in a noticeably long time span between a new idea/concept is validated until it is implemented in products and applications as part of a worldwide standard. Each ISO/IEC MPEG standard can be considered as a "frozen" version or a snapshot of state-of-the-art of video compression taken a few years before the standard is released in its final form to the public. This delay is a significant impediment to providing innovations to the market, thus opening the way to ad-hoc proprietary solutions that jeopardize all efforts made by standardization bodies to ensure interoperability and openness to state of the art technology to all operators.

Considering all these problems and drawbacks of the process used so far for developing a new standard, MPEG is currently finalizing a new (standard) framework called *Reconfigurable Video Coding* (RVC). The goal of the MPEG RVC effort is to address the limitations and problems presented above and thus offer a more flexible use and faster path to innovation of MPEG standards in a way that is competitive in the current dynamic environment. In order to achieve these goals, a paradigm shift for both specification formalism and for the concept of a standard video codec itself is necessary. The new RVC standard is based on building a unified library of video coding algorithms. So what become standard is the "module" of a library that can be updated incrementally as soon as new video technology is demonstrated to be valuable. A RVC codec is specified as a configuration of modules (called in MPEG Functional Units), using an asynchronous data flow computation model. Each module behavior is specified in RVC-CAL (an actor data flow language) and the connections by and XML dialect called Functional unit Network Language (FNL). All these languages are standardized by MPEG with the objective of, on one side providing all functionality an expressiveness appropriate for the RVC specification goals (modularity, compactness, expressiveness) and on the other side to be supported by efficient tools that directly synthesize both SW and HW implementations. In fact the new RVC specification formalisms obviously provides advantages even when employing the classical methods based on hand writing SW code or HDL code, however the possibility of generating such implementations by tool synthesis is certainly much more attractive. Therefore the MPEG RVC effort has the ambition to provide a more flexible way of providing standard video technology using a specification formalism laying at a higher level of abstraction than the one traditionally used, but at the same time provide a starting point for implementation that is compatible with more efficient methodologies for developing implementation for the next generation of multicore and heterogeneous platforms. It can be noticed that the effort of raising the abstraction level and unifying a specification of both SW and HW is so far experimented and tuned in the video coding field, but has certainly a potential for a much wider usage in other signal processing and communication application fields.

This paper introduces the components and new concepts of the new MPEG RVC framework and then focuses on the developments, challenges and initial results yielded by the synthesis tools that directly generate both SW and HW implementations from dataflow RVC specifications. The paper continues with Sec. 2 with an introduction to the essential concepts of the data flow approach and the basic elements of the CAL dataflow language. Then Sec. 3 introduces and describes the other different standard components of the MPEG RVC framework: the languages for the description of new codec configurations and of the corresponding bitstream syntax, the libraries of the standard video coding algorithms and the instantiation of the behavioral model of the decoder. Section 4 focuses on the non normative tools that support the development of the MPEG RVC specifications and their implementations. It describes the tools that supports simulation of the behavioral model and particularly all tools and their backhands that provides direct synthesis of the standard RVC specification into SW and HW implementations. Section 5 reports some results of the implementations yielded by the new tools and discusses further optimizations that could be employed to improve the tool efficiency. Section 6 concludes the paper by summarizing the results achieved so far and presenting some ideas for future works and further developments.

2. DATAFLOW - PROGRAMMING WITH STREAMS

The form of *dataflow* that we are concerned with in this work is best viewed as dealing with *streams*, i.e. (possibly unbounded) sequences of data objects called *tokens*. A dataflow program is defined as a directed graph, where the nodes represent computational units, called *actors*, and the arcs represent the flow of data, a token stream flowing along each arc.

Actors have input and output *ports* through which they receive and send token from and to their environment. Each connecting arc leads from an output port to an input port. The flow of tokens along those arcs is lossless and order-preserving: every token that the sending actor emits to the output port is guaranteed to arrive at the connected input port (of another actor or the same actor in the case of direct feedback), and tokens arrive in exactly the order in which they were sent. No guarantees are made beyond this, in particular not with respect to timing, which the dataflow model abstracts from. Consequently, actors send and receive tokens in relative asynchrony, and tokens may be buffered on the way from the sender to the receiver.

2.1 Actors

Actors are the basic computational entities of a dataflow program. As in [7], the actors in our model execute by performing a number of discrete computational steps, also referred to as *firings*. During each firing, an actor may:

- consume tokens from its input ports,
- produce tokens on its output ports,
- modify its internal state (if it has any).

An actor may contain memory that it uses to store local state. An important guarantee of the actor model is that this state not be shared with other actors, i.e. actors communicate with one another exclusively through passing tokens along dataflow connections, and not through shared state. This eliminates race conditions between actors, and makes dataflow programs more resilient to the influences of different scheduling policies.

A firing, once initiated, must terminate irrespective of the environment of the actor, i.e. all conditions necessary for its termination (such as the presence of sufficient input tokens) must be ascertained before the firing is begun. Between firings, the actor is in quiescence, i.e it does not change its state, does not consume tokens, and does not produce tokens.

2.2 The CAL Actor Language

CAL [5] is a for writing actors. It has been used in a wide variety of applications and has been compiled to hardware and software implementations, and work on mixed HW/SW implementations is under way. CAL represents the basic components of a dataflow actor with firing in a straightforward manner, providing structuring mechanisms that help the user to understand the functioning of an actor, and that aid tools to extract relevant information from an actor description at compile time.

A simple example of a CAL actor is shown in the `Add` actor below, which has two input ports `t1` and `t2`, and one output port `s`, all of type `int`. The actor contains one *action* that consumes one token on each input ports, and produces one token on the output port. Actions define what happens during the firing of an actor, and also when a firing may occur. In this case, the declaration of input token variables implies that there be one token on both ports `t1` and `t2`.

```
actor Add () int t1, int t2 ⇒ int s :
  action [a], [b] ⇒ [a + b]  end
end
```

Actors may include state variables, such as the variable `sum` in the `Sum` actor below. If they do, then actions may modify those state variables in their *bodies*, which are fragments of conventional imperative code between the `do` and `end` keywords:

```
actor Sum () int t ⇒ int s :
  int sum := 0;

  action [a] ⇒ [sum]
  do
    sum := sum + a;
  end
end
```

An actor may have any number of actions. The `Merge` actor below has two, each of which copies a token from one of the two inputs to the output port.

```
actor Merge () int Input1, int Input2 ⇒ int Output:
  action Input1: [x] ⇒ [x] end
  action Input2: [x] ⇒ [x] end
end
```

Should a token be present on both input ports, either action may fire. The actor above does not prescribe any policy for choosing between the actions should they both be firable, and consequently its result is non-deterministic. While occasionally useful, non-determinism is a property that actor writers need to be aware of and able to control, which is why many constructs of the CAL language deal with ways to specify action selection and constraining action firability, and they permit tools to detect potential non-determinism.

One mechanism for picking actions is to use *guards*, conditional expressions associated with an action. The `Select` actor below reads and forwards a token from either port `A` or `B`, depending on the value of the token read from the `S` input port. That value is tested in the guards of the actions.

```
actor Select () boolean S, int A, int B ⇒ int Output:

    action S: [sel], A: [v] ⇒ [v]
    guard sel end

    action S: [sel], B: [v] ⇒ [v]
    guard not sel end
end
```

In general, guards are arbitrary boolean expressions that may refer to input tokens and state variables.

Another way to control action selection is to order action with respect to their *priority*. In the `BiasedMerge` actor a priority order is established between the two actions labeled `A` and `B`. It ensures that in case both actions can otherwise fire, the one labeled `A` will be given preference over the one labeled `B`.

```
actor BiasedMerge () int Input1, int Input2 ⇒
int Output:
  A: action Input1: [x] ⇒ [x] end
  B: action Input2: [x] ⇒ [x] end
```

```
  priority A > B; end
end
```

For an in-depth description of the language, the reader is referred to the language report [5]. A large selection of example actors is available at the OpenDF repository,[1] among them the MPEG-4 decoder discussed below.

3. RECONFIGURABLE VIDEO CODING STANDARD

3.1 Requirements

The initial investigations for the definition of the MPEG Reconfigurable Video Coding (RVC) framework [2, 1] started in 2004. The new MPEG RVC (ISO) standard is currently under its final stages of standardization (see Fig. 1). From a standardization point of view the new concept behind RVC is to unify the specification of existing standards by using a common library of components, called in RVC Functional Units (FUs), and to be also able to specify completely new configurations that may better satisfy application-specific constraints by selecting standard components from a library of standard coding algorithms. The reasons that lead MPEG to search for such a new approach for the specification of video compression algorithms have been presented in details in Sec 1. Such motivations were translated into a set of requirements that the new formalism should have satisfied [14, 2]. As described above the encapsulation capabilities of CAL actors satisfy the requirement of developing a library of modules that encapsulate the essence of re-usable algorithms, the data dependencies are made explicit by the data token exchanges at input and output ports, and the the asynchronous behavior of a network of actors abstracts from time so no specific scheduling, beside the ones constrained by the intrinsic data dependencies, is unnecessarily provided as standard specification, but any scheduling compatible with the computation model and appropriate for each specific implementation could be selected. Therefore, the selection of CAL networks of actors as base formalism for MPEG RVC resulted as the choice better satisfying the fundamental requirements set by the committee in the investigation phase. It can be noticed that in principle any common imperative language could have been used to implement the desired features, for instance by appropriate C++ classes and methods, however the final code would have resulted of more difficult readability, less compact and conformance to the standard should have relied on the correct "constrained usage" of the language and of such specific constructs by the users. Conditions that would have been very difficult or impossible to enforce. The choice thus was to standardize specific (sub)-sets with a few necessary extensions of existing or new languages for each required specification functionality. In fact the possibility of dynamic configuration and reconfiguration of codecs also requires new methodologies and new tools for describing the new bitstream syntaxes and the parsers of such new codecs

3.2 Description of the standard components

Two ISO standards are defined so far within the MPEG RVC framework: ISO/IEC 23001-4 [11]) (also called MPEG-B part 4) and ISO/IEC 23002-4 [12]) (MPEG-C part 4). International Standard ISO/IEC 23001-4 defines the components of the overall framework as well as the standard languages that are used to specify an existing or a new codec configuration of a RVC decoder. The essential component of the normative specification of a codec configuration is the dataflow description called in RVC the "Abstract Decoder Model" (ADM). It is an executable description that specify the overall input output behavior that all implementations derived by such specification need to satisfy. So as to build such ADM two normative inputs are necessary: the decoder description and the and the standard library of components. The Decoder Description (Fig. 2) includes two components:

- **The Bitstream Syntax Description (BSD)**, specified by the RVC Bitstream Syntax Description Language (RVC-BSDL), a language syntactically describing the structure of the input encoded bitstream which is a subset of the standard MPEG Bitstream Syntax Description Language (ISO/IEC 23001-4). Such description is used to specify the appropriate parser to decode the corresponding input encoded data [13, 23].
- **The FU Network Description (FND)**, which describes the structure of the decoder (i.e. the instantiation of the library components (FUs)) and their connections. It also specifies the values of the parameters used for each instantiation of the different FUs composing the decoder [4, 15, 24]. The FND is written in the so called FU Network Language (FNL).

The syntax parser (built from the BSD), together with the network of FUs (built using the FND and the instantiation of the FUs from the RVC library), form the Abstract Decoder Model (ADM), which finally constitutes the normative, in the ISO/MPEG meaning, behavioral model of the decoder. In summary the International Standard ISO/IEC 23001-4 defines/stamndardizes three new languages used to specify the RVC ADM:

- RVC-CAL, a subset of CAL language operators and constructs including a set of data types, used to specify the RVC library of video coding algorithms or FUs,
- FNL, the language describing the network of FUs, essentialli an XML dialect
- RVC-BSDL, a subset of the standard MPEG BSDL with a few extensions, used to describe the bistream syntax and implicitly the parsers of a new codec configuration

The main reason for standardizing subsets, with a few minor extensions, of languages that could include more operators and constructs was to facilitate the design of efficient synthesis tools that could translate RVC-CAL and RVC-BSDL descriptions into respectively SW and HW implementations and RVC-CAL parsers respectively. The fact that such languages are fully supported by tools is an onvious Obviously the restrictions imposed The International Standard ISO/IEC 23002-4 specifies the textual description of the unified library of video coding algorithms employed in the current MPEG standards. Up to now, two MPEG standards/profiles are fully covered:

- MPEG-4 part 2 Simple Profile,
- MPEG-4 part 10 (AVC) Constrained Baseline Profile.

[1] `http://www.opendf.net`

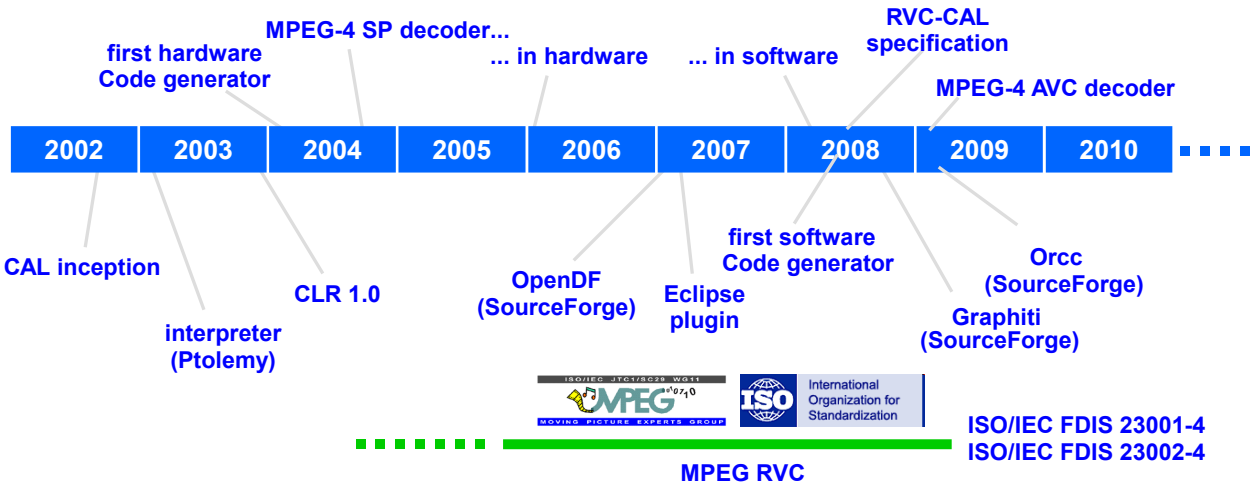

Figure 1: Pictorial representation of CAL language and tools development and the timeline of the steps of MPEG RVC standardization.

Several extensions are in development. Amendment 1 of ISO/IEC 23002-4 will include the conformance testing procedure for the FU as well as the reference software written in RVC-CAL of the RVC MPEG toolbox. It is planned to be finally completed by Q1 of 2010. Amendment 2 of ISO/IEC23002-4, currently in advanced development, will include MPEG-4 AVC High Profile (FREXT profile), MPEG-4 SVC baseline profile, MPEG-4 Part 2 Advanced Simple profile and MPEG-2 Main Profile. The mechanisms for the transport of RVC codec descriptions and bitstream syntax descriptions are currently under core experiment stage. Various scenarios enabling downloads and dynamic update of codec configurations on processing platform are analyzed so as to verify what (if any) amendment to MPEG-2 and MPEG-4 Systems standard [9, 10] are needed to support the widest class of deployment scenarios for RVC codecs.

3.3 The toolbox library

The innovative feature of the RVC standard that distinguishes it from traditional monolithic specification of video coding standards is that video technology is unified into a single library and the traditional concept of conformance to a given MPEG standard need to be updated to the new scenario. Figure 3 illustrates this conceptual view of conformance to the RVC framework [21]. All the three types of decoders are constructed and specified using the languages standardized in MPEG-B. Hence, their normative specification conform to the MPEG-B standard. A `Type-1` decoder is constructed using the FUs within the MPEG VTL only. Hence, this type of decoder conforms to both the MPEG-B and MPEG-C standards. A `Type-2` decoder is constructed using FUs from the MPEG VTL as well as one or more proprietary libraries (VTL 1-n). This type of decoder conforms to the MPEG-B standard only. Finally, A `Type-3` decoder is constructed using one or more proprietary VTL (VTL 1-n), without using the MPEG VTL. This type of decoder also conforms to the MPEG-B standard only. An RVC decoder (i.e. conformant to MPEG-B) is composed of coding tools described in VTLs according to the decoder description. The MPEG VTL is described by MPEG-C. The MPEG VTL is normatively specified using RVC-CAL.

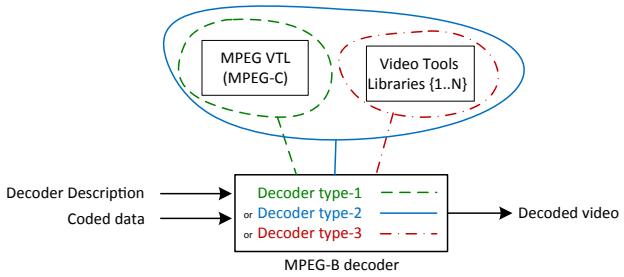

Figure 3: The conceptual view of RVC.

An appropriate level of granularity for the components of the unified standard library is important, to enable an effective possibility of reconfigurations, for codecs, and an efficient reuse of components in codec implementations. If the library would have been composed of too coarse modules, they would have resulted too large to enable their usage in different and useful codec configurations, whereas, if library component granularity level would have been too fine, the number of modules in the library would have resulted too large for an efficient and practical reconfiguration process at the codec implementation side, and could have obscured the desired high-level description and modeling features of the RVC codec specifications. Most of the efforts behind the development and standardization of the MPEG VTL were devoted to study the best granularity trade-off level of the VTL components. However, it must be noticed that the choice of the best trade-off in terms of high-level description and module re-usability, does not affect the potential parallelism of the algorithms that can be exploited in heterogeneous, multi-core and FPGA implementations by the usage of proprietary libraries.

3.4 Instantiation process of a RVC ADM

As discussed above the process of generating a decoder im-

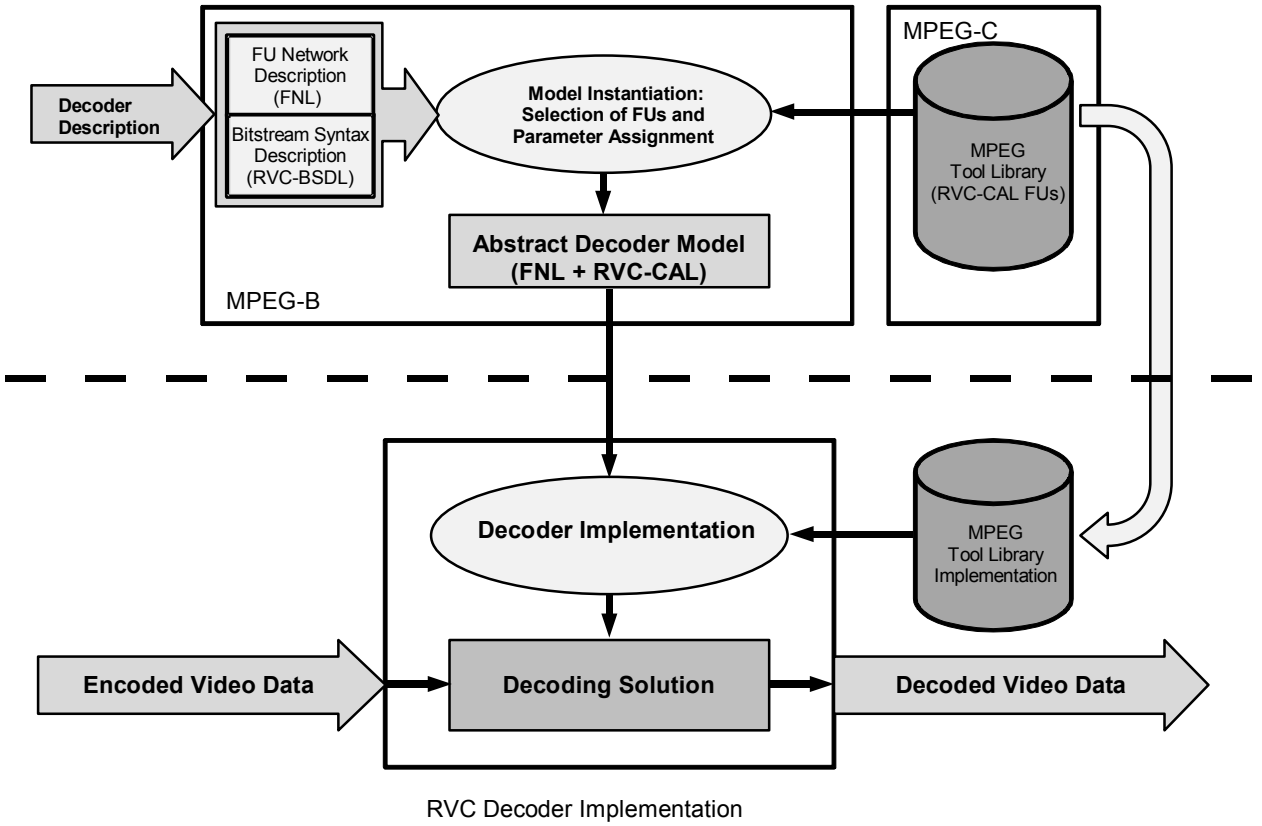

Figure 2: Graphic representation of the components of the RVC standard. On top the normative descriptions that are used to build the ADM. On bottom any non normative process that yields a conformant decoder implementation of the normative ADM.

plementation on a decoding platform implies the acquisition of a Decoder Description that fully specifies the architecture of the decoder and the structure of the incoming bitstream. So as to instantiate the corresponding decoder implementation, the decoding platform uses a library of building blocks specified by MPEG-C. Conceptually, such library is a user defined proprietary implementation of the MPEG RVC standard library, providing the same I/O behavior for each component. Such library can be expressly developed to explicitly expose an additional level of concurrency and parallelism appropriate for implementing a new decoder configuration on the user specific heterogeneous/multi-core target platforms. The dataflow form of the standard RVC specification, with the associated Model of Computation, guarantee that any reconfiguration of the user defined proprietary library, developed at whatever lower level of granularity, provides an implementation that is consistent with the (abstract) RVC decoder model that is originally specified using the standard library. Figure 2 and 3 show how a decoding solution is built from, not only the standard specification of the codecs in RVC-CAL by using the normative VTL, and this already provides an explicit, concurrent and parallel model, but also from any non-normative "platform-friendly" proprietary Video Tool Libraries, that increases if necessary the level of explicit concurrency and parallelism for specific target platforms. Thus, the standard RVC specification, that is already an explicit model for concurrent systems, can be further improved or specialized by proprietary libraries that can be used in the instantiation phase of an RVC codec implementation.

3.5 Commonalities between decoders

All existing MPEG codecs are based on the same architecture, the hybrid decoding architecture including a parser that extracts values for texture reconstruction and motion compensation. Indeed, MPEG-4 SP and MPEG-4 AVC are typical examples of hybrid decoders. Figure 4 shows the main functional blocks composing an hybrid decoder architecture.

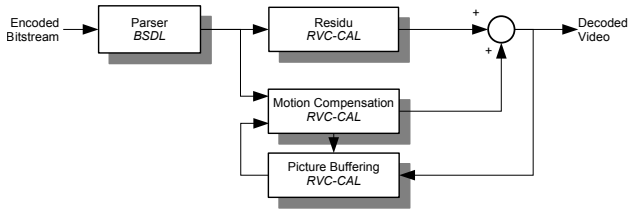

Figure 4: Hybrid decoder structure

As mentioned earlier, a RVC decoder is described as a block diagram with FNL [11], an XML dialect that describes the network of interconnected actors instantiated from the Standard MPEG tool library. Basing on the current finalized standards, results regarding two RVC design cases can be reported by the works of RVC RVC experts [26, 15]. They are the specifications of MPEG-4 Simple Profile decoder and MPEG-4 AVC Constrained Baseline Profile decoder [6], both of them entirely written in RVC-CAL.

4. TOOLS

The fact that a new standard provides a streaming data flow based specification of a complex processing system such as a video decoder and that such specification has "nice" features and is a good starting point for various form of concurrent implementations, it does not mean that all implementation challenges are solved. Many methodologies can be applied to a RVC ADM to yield a final implementation including the "old style" handwriting in whatever implementation language of the RVC ADM. However, the more attracting approaches are the ones that employ tools for the direct synthesis of SW and HW implementations. The rest of the paper focuses on the tools that support such implementation approach reporting the tools developed so far, the results obtained and discussing the paths, challenges and optimizations that currently remain open and active topics for research.

4.1 Related Work and Tools

Since the behavior of the ADM is the conformance reference of any RVC specification, a simulator is needed for the validation of any RVC specification. RVC-CAL is currently supported by a portable interpreter infrastructure that can simulate a network of actors (i.e any RVC-ADM). Such interpreter was first developed in the Moses project [2]. Moses features a graphical network editor, and allows the user to monitor actor execution (actor state and token values). The project and related simulator, being no longer continued and maintained, has been superseded by two plugins in the Eclipse environment: the Open Dataflow environment (OpenDF [3] [3]) for editing CAL actor and the Graphiti editor for graphically editing networks. CAL programs (i.e. network of actors) can also be simulated in the Ptolemy II [4] environment.

4.2 RVC-CAL Compiler Infrastructure

Once the ADM is validated by simulations the following step is to develop a conformant implementation. If the target platform is SW, a compiler of RVC-CAL is the necessary tool. A common compiler framework called Orcc (Open RVC-CAL Compiler) has been designed and developed to support the synthesis of SW codes from CAL language. Orcc is the successor of a first version of software code generator called Cal2C and described in [26]. Cal2C translates RVC-CAL actors to C and uses SystemC [8], a uniprocessor simulation framework, to implement the networks and to create a scheduler for the actors. Cal2C has the following limitations:

1. it is a monolithic compiler
2. it can only generate C code for each actor translation
3. it can only generate SystemC to schedule actor executions
4. it is not extensible.

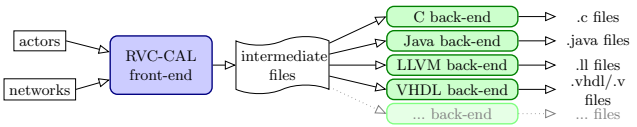

Figure 5: Open RVC-CAL Compiler Infrastructure.

Orcc overcomes such limitations by having been structured into different components: one front-end and several back-ends, as illustrated in Fig. 5. The compilation process transforming a given RVC-CAL dataflow program into a target language is a two-step process. First, the front-end (1) parses the given networks, possibly hierarchical networks, (2) flattens the network hierarchy and represents the result as a "flat" network, (3) parses actors and translates them to an Intermediate Representation (IR), (4) serializes each actor in IR form to a file. Then, the appropriate back-end loads the flat network and actors representations, in IR form, and generates code in the target language.

Orcc is provided as an Eclipse feature composed of two plugins. Eclipse is a development environment as well as an open extensible application framework upon which software can be built [25]. The first "main" plugin allows back-end implementations to read and write intermediate files in memory and to manipulate the IR. It includes common transformations, such as instantiation of actors and propagation of constant parameters.

The second "back-ends" plugin has currently five functional built-in implementations of back-ends: C, C++, Java, VHDL/verilog and LLVM. Available back-ends produce software and hardware codes. The hardware code generator presented in more details in [15] is part of Orcc, it generates a hardware description from CAL by translating Orcc's intermediate representation to a lower-level IR called XLIM, and then compiling XLIM to a Hardware Description Language (HDL). In future Orcc would probably evolve proving a functionality directly generating a Verilog or VHDL back-end from its own intermediate internal representation.

4.3 Actor and Network Code Generation

Given a specific target language, actor code generation follows the steps shown on Fig. 6. The IR is parsed and represented in-memory as instances of classes. Such classes support the visitor design pattern, which makes writing transformations of the IR easier. Generic transformations can be applied to the IR, depending on the back-end used. For instance a generic transformation used by non-SSA back-ends is a naive "out of SSA" translation, which allocates one variable per SSA register and replaces ϕ assignments by copies. The next step optionally extends the IR by adding back-end-specific instructions where needed or desirable. Back-end-specific instructions are increment/decrement for C/C++, `zext` (zero extends) for LLVM, etc. The specific IR may then undergo specific transformations. An example of a specific transformation is a transformation in the LLVM back-end

[2]Moses project: http://www.tik.ethz.ch/ moses/
[3]Open Dataflow: http://opendf.sf.net/
[4]Ptolemy II: http://ptolemy.eecs.berkeley.edu/

that converts expressions to LLVM instructions. Finally, the specific IR is translated to LLVM code.

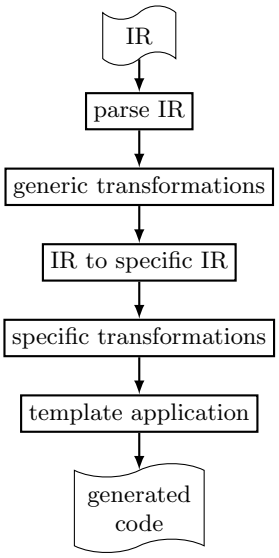

Figure 6: Actor code generation steps

Network code generation is by far easier than actor code generation. The network name, the list of broadcasts, instances, and connections in the network, are passed to the underlying template as attributes. The template is then called to produce text as a result.

Orcc considers that all actors have data-dependent firing conditions, and that the Dataflow Process Networks (DPN) [20] model must be used. A DPN contains actors that communicate with each other using unidirectional FIFOs, where reads are blocking and writes are nonblocking. DPNs must be scheduled dynamically, hence actors are scheduled at runtime by an *actor scheduler*. In [26], Wipliez et al. implemented a SystemC actor scheduler, but recommended a new scheduler be developed avoiding to use threads.

The scheduler recently developed is a uniprocessor simple round-robin scheduler, yet it is very efficient compared to the SystemC scheduler. The scheduler endlessly schedules all actors one after the other according to an arbitrary order, which in this case is simply the alphabetical order.

In the current version of the code generators, each actor is translated separately and is connected with FIFO buffers. Consequently, no cross-actor optimizations are employed at the current level of development of the tool. If two actors connected in this manner are specified to belong to different clock domains, an asynchronous FIFO implementation is selected, otherwise a synchronous FIFO is used for compactness of the implementation. Actors interact by means of FIFOs using a handshake protocol, which allows them to detect when a data token is available or when a FIFO is full.

4.4 C-based Back-ends

Orcc comes with four built-in back-ends that use the same specific IR: C, C++, Java and VHDL/verilog. The specific IR only adds increment/decrement instructions and self-assignment instructions. A self-assignment is an instruction such as

```
x *= 3;
```

instead of the vanilla IR representation

```
tmp1 := load(x);
store(x, tmp1 * 3);
```

While not strictly necessary, these instructions makes the output code look more similar to C.

The essential differences between the three following back-ends C, C++, and Java are: (1) the generic transformations that are applied, (2) the templates. Namely the C back-end implementation of the `write` operation is that it merely returns a pointer to the FIFO and advances the number of tokens written in the FIFO: The `write` operations that may be present in actions thus need to be moved *before* any `store` done to the FIFO. Conversely, the C++ and Java back-ends currently copy contents loaded from/stored to FIFOs and do not need such a transformation. The rest of the differences between these three languages is done at the template level. Note that the C++ back-end is slightly more complex than the others as it produces one header file and one implementation file per actor by using one template for each.

4.5 LLVM back-end

The key idea behind the development of an LLVM back-end has two objectives. Since Orcc IR is closer to the LLVM IR, thus generating LLVM directly from Orcc prevents a loss of information that would occur when instead generating code using the C-based back-ends. Both Orcc IR and the LLVM IR are in SSA form with an unlimited number of registers, whereas C or Java need to allocate every variable on the stack. Both IRs have an integer type with an arbitrary bit width, whereas most languages only support fixed size integer types. Both IRs have instructions with similar semantics, those include assignment to a local variable, load/store memory operations, ϕ assignments.

The main difference between LLVM and Orcc IR is that conditional branch nodes, namely `if` and `while`, have no direct equivalent in LLVM. The Control Flow Graph (CFG) of a function in the LLVM IR is a list of basic blocks, each basic block starting with a label. A basic block contains a list of instructions, and ends with a terminator instruction, such as a branch instruction or function return instruction. As a result, the actor IR is augmented with LLVM-specific nodes by a transformation pass that simplifies, flattens the CFG, and adds label nodes and branch nodes. We believe that Orcc IR needs to be improved by supporting labels on control flow nodes if we want to remove such LLVM-specific nodes.

A second difference between LLVM and the actor IR in Orcc is the fact that Orcc IR supports arithmetic expressions in assignments, load/store, and conditional branches, while LLVM only has support for three-address code(3AC) [18]. Therefore, every expressions containing more than one fundamental operation in Orcc IR is translated into a series of three-address code instructions before generating the LLVM IR.

The LLVM back-end allows users to take advantage of the LLVM infrastructure, surrounding off-line compilers for X86, X86-64, PowerPC 32/64, ARM, Thumb, IA-64, Alpha, SPARC, MIPS and CellSPU architectures, as well as a Just-In-Time compiler for X86, X86-64, PowerPC 32/64 processors. As LLVM is becoming a commercial grade research compiler, the code generated by the LLVM back-end will continually benefit from improvements brought to LLVM

implementations.

4.6 VHDL/Verilog back-end

In case the target platform for the implementation of a RVC ADM is based on reconfigurable or hardwired HW technology a back-end for the generation of HDL is needed. A first step of the HDL generation is based in language transformation to obtain simpler language constructs. Then a precompilation stage follows, Orcc performs basic source code transformations so as to translate the actor structure in a form more amenable to hardware implementations, such as e.g. inlining procedures and function calls. Then the canonical, closed actors are translated into a collection of communicating threads.

In the current back-end implementation, an actor with N actions is translated into $N + 1$ threads, one for each action and another one for the *action scheduler*. The action scheduler is the mechanism that determines which action to fire next, based on the availability of tokens, the guard expression of each action (if present), the finite state machine schedule, and action priorities.

The final phase of the translation process generates an RTL implementation (in Verilog) from a set of threads in SSA form. The first step simply substitutes operators in expressions for hardware operators, creates the hardware structures required to implement the control flow elements (loops, if-then-else statements), and also generates the appropriate muxing/demuxing logic for variable accesses, including the ϕ elements in the SSA form.

The resulting basic circuit is then optimized in a sequence of steps.

1. **Bit-accurate constant propagation.** This step eliminates constant or redundant bits throughout the circuit, along with all wires transmitting them. Any part of the circuit that does not contribute to the result is also removed, which roughly corresponds to dead code elimination in traditional software compilation.

2. **Static scheduling of operators.** By default, operators and control elements interact using a protocol of explicit activation—e.g., a multiplier will get triggered by explicit signals signifying that both its operands are available, and will in turn emit such a signal to downstream operators once it has completed multiplication. In many cases, operators with known execution times can be scheduled statically, thus removing the need for explicit activation and the associated control logic. In case operands arrive with constant time difference, a fixed small number of registers can be inserted into the path of the operand that arrives earlier.

3. **Memory access optimizations.** Arrays are mapped to Block RAMs (BRAM) for FPGA implementation. These usually small RAM blocks (typically 18 kBits) are distributed across the FPGA, and can be ganged up to form larger memories, or a number of small arrays may be placed into one BRAM. Furthermore, BRAMs usually provide two or more ports, which allows for concurrent accesses to the same memory region. Based on an analysis of the sizes of arrays and the access patterns, the backend maps array variables to Block RAMs, and accesses to specific ports.

4. **Pipelining, retiming.** In order to achieve a desired clock rate, it may be necessary to add registers to the generated circuit in order to break combinatorial paths, and to give synthesis back-ends more opportunity for retiming.

Currently only Verilog generation is supported, but there is no reason for which a VHDL generator back-end cannot be developed using the same approach.

5. CASE STUDIES

5.1 RVC MPEG-4 Simple Profile Decoder description

Figure 7 reports a network representation of the RVC MPEG-4 Simple Profile (SP) decoder description. The parser is a hierarchical network of actors (each of them is described in a separate FNL file). All other blocks are actors written in RVC-CAL. Figure 7 presents the structure of the MPEG-4 SP ADM as described within MPEG RVC. Essentially it is composed of four mains parts: the parser, a luminance component (Y) processing path, and two chrominance components (U, V) processing paths. Each of the path is composed by its texture decoding processing stage as well as its motion compensation stage (both are hierarchical RVC-CAL networks of Functional Units).

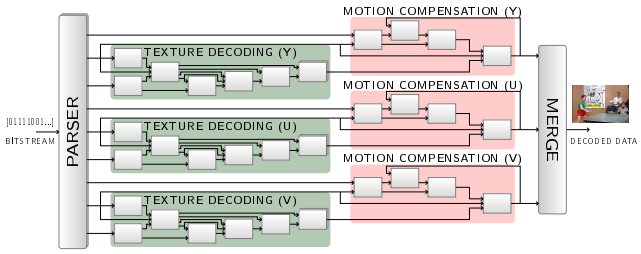

Figure 7: MPEG-4 Simple Profile decoder description

The MPEG-4 SP ADM obviously results to be a dataflow program (Figure 7, Table 3) that is is composed of 27 atomic FUs (or actors in dataflow programming) and 9 subnetworks (actor/network composition). Single atomic actors can be instantiated several times, for instance there are 42 actor instantiations in this dataflow program. Figure 8 shows a top-level view of the decoder. The main functional blocks include the bitstream parser, the reconstruction block, the 2-D inverse cosine transform, the frame buffer and the motion compensation module. These functional units are themselves hierarchical compositions of actor networks.

5.2 HW synthesis

Some of the authors have performed an implementation study [15], in which the RVC MPEG-4 SP decoder specified in RVC-CAL according to the MPEG RVC formalism has been implemented on an FPGA using a CAL-to-HDL code generator. The objective of the design case was to support 30 frames of 1080p in the YUV420 format per second, which amounts to a production of 93.3 Mbyte of video output per second. The given target clock rate of 120 MHz implies 1.29 cycles of processing per output sample on average. The

results of the design case were encouraging in that the code generated from the MPEG RVC-CAL specification did not only outperformed the handwritten reference in VHDL, both in terms of throughput and silicon area, but also allowed for a significantly reduced development effort. Table 1 shows the comparison between CAL specification and the VHDL reference implemented over a Xilinx Virtex 2 pro FPGA running at 100MHz.

It should be emphasized that this counter-intuitive result cannot be attributed to the sophistication of the HDL synthesis tool. On the contrary the tool does not perform a number of potential optimizations, such as for instance optimizations involving more than one actor. Instead, the good results appear to be yield by the implementation and development process itself. The implementation approach was based generating a proprietary implementation of the standard MPEG RVC toolbox composed of FUs of lower level of granularity. Thus the implementation methodology was to substitute the FU of the standard abstract decoder model of the MPEG-4 SP with an equivalent hierarchical implementation, in terms of behavior. Essentially standard toolbox FU were substituted with networks of FU described as actors of lower granularity. A notable difference of such implementation approach when compared with the classical hand writing of HDL code from a textual or sequential specification (i.e. a C/C++ program for instance) was that the CAL specification of the proprietary implementation toolbox (that can be directly derived from the standard RVC toolbox) could go through significantly more design iterations than the one applicable from a handwritten VHDL reference —in spite of being developed in approximately a quarter of the development time (including the time of developing the standard MPEG RVC toolbox from scratch). Whereas a dominant part of the development of a classical VHDL reference development need to be spent getting the system to work correctly, the effort of the CAL specification could be focused on optimizing system performance to meet the design constraints.

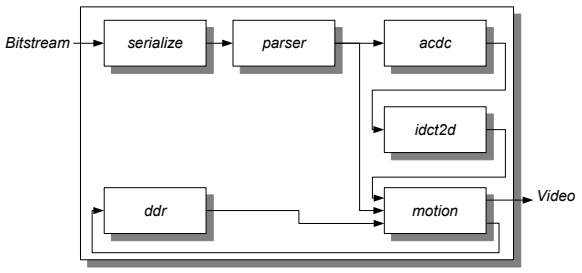

Figure 8: Top-level dataflow graph of the proprietary implementation of the RVC MPEG-4 decoder.

The initial design cycle of the proprietary RVC library resulted in an implementation that was not only inferior to the VHDL reference, but one that also failed to meet the throughput and area constraints. Subsequent iterations explored several other points in the design space until arriving at a solution that satisfied the constraints. At least for the considered implementation study, the benefit of short design cycles seem to outweigh the inefficiencies that resulted from high-level synthesis and the reduced control over implementation details.

	Size slices BRAM	**Speed** kMB/s	**Code Size** kSLOC	**Dev. time** MM
CAL	3872, 22	290	4	3
VHDL	4637, 26	180	15	12
Improv. factor	1.2	1.6	3.75	4

kMB/s=kilo macroblocks per second
kSLOC=kilo source lines of code

Table 1: Hardware synthesis results for a proprietary implementation of a MPEG-4 Simple Profile decoder. The numbers are compared with a reference hand written design in VHDL.

In particular, the asynchrony of the programming model and its realization in hardware allowed for convenient experiments with design ideas. Local changes, involving only one or a few actors, do not break the rest of the system in spite of a significantly modified temporal behavior. In contrast, any design methodology that relies on precise specification of timing —such as RTL, where designers specify behavior cycle-by-cycle— would have resulted in changes that propagate through the design.

Table 1 shows the quality of result produced by the RTL synthesis engine of the MPEG-4 SP video decoder. Note that the code generated from the high-level dataflow RVC description and proprietary implementation of the MPEG toolbox actually outperforms the hand-written VHDL design in terms of both throughput and silicon area for a FPGA implementation.

5.3 SW synthesis

Direct C-synthesis [24, 26] of the ADM specification of the MPEG-4 SP provided by the RVC standard (Figure 7)or in other words the dataflow program of a MPEG-4 SP decoder is another interesting case study for evaluating the efficiency of the methodology. The SW code generator, also presented in more details in [24], uses process network model of computation [16] to implement the CAL dataflow model. The compiler creates a multi-thread program from the given dataflow model, where each actor is translated into a thread and the connectivity between actors is implemented via software FIFOs. Although the generation provides correct SW implementations, inherent context switches occur during execution, due to the concurrent execution of threads, that may lead to inefficient SW execution if the granularity of actor is too fine. The problem of multi-threaded programs is discussed in [19]. A more appropriate solution that avoids thread management are presented in [20, 22]. Instead of suspending and resuming threads based on the blocking read semantic of process network [17], actors are, instead, managed by a user-level scheduler that select the sequence of actor firing. The scheduler checks, before executing an actor, if it can fire, depending on the availability of tokens on inputs and the availability of rooms on outputs. If the actor can fire, it is executed (these two steps refers to the *enabling function* and the *invoking function* of [22]). If the actor cannot fire, the scheduler simply tests the next actor to fire (sorted

following an appropriate given strategy) and so on. A code generator based on such concept [26] is available at Orcc website [5]. Such compiler presents a scheduler that has the two following characteristics: (1) actor firings are checked at run-time (the dataflow model is not scheduled statically), (2) the scheduler executes actors following a round-robin strategy (actors are sorted a priori). For instance, in the case of the standard RVC MPEG-4 SP dataflow model such generated mono-thread implementation is about 4 times faster than the one obtainable by [24]. Table 2 shows that synthesized C-software is faster than the simulated CAL dataflow program (80 frames/s instead of 0.15 frames/s), and twice the real-time decoding for a QCIF format (25 frames/s). However it remains much slower than the automatically synthesized hardware description by Cal2HDL [15].

MPEG4 SP decoder	Speed kMB/s	Clock speed GHz	Code size kSLOC
CAL simulator	0.015	2.5	3.4
Cal2C	8	2.5	10.4
Cal2HDL	290	0.12	4

Table 2: MPEG-4 Simple Profile decoder speed and SLOC.

As mentioned above, the MPEG-4 SP dataflow program is composed of 61 actor instantiations in the flattened dataflow program. The flattened network becomes a C file that currently contains a round robin scheduler for the actor scheduling and FIFOs connections between actors. Each actor becomes a C file containing all its action/processing with its overall action scheduling/control. Its number of SLOC is shown in Table 3. All of the generated files are successfully compiled by gcc. For instance, the "ParserHeader" actor inside the "Parser" network is the most complex actor with multiple actions. The translated C-file (with actions and state variables) includes 2062 SLOC for both actions and action scheduling. As a comparison, the original CAL file contains 962 lines of codes.

MPEG-4 SP decoder	CAL	C actors	C scheduler
Number of files	27	61	1
Code Size (kSLOC)	2.9	19	2

Table 3: Code size and number of files automatically generated for MPEG-4 Simple Profile decoder.

A comparison of the RVC-CAL description (Tab. 4) shows that the MPEG-4 AVC decoder is twice more complex in RVC-CAL than the MPEG-4 SP RVC-CAL description. Some components of the model have already been redesigned so as to improve pipelining and parallelism among actors. A simulation of the MPEG-4 AVC RVC-CAL model on a *Intel Core 2 Duo* @ 2.5Ghz is more than 2.5 slower than the RVC MPEG-4 SP description.

Comparing to the MPEG-4 SP CAL model, the MPEG-4 AVC decoder has been developed exploiting more CAL optimization possibilities (for instance processing of several tokens in one firing) while remaining fully conformant with the RVC-CAL standard. Thanks to the relevant complexity of a MPEG-4 AVC RVC-CAL description, the direct synthesis of a SW implementation is a meaningful way to test the efficiency of the current RVC support tools. The performances of the current SW code generation of the MPEG-4 AVC decoder are quite promising since they can achieve up to 53 fps, considering that no inter-actor and platform specific optimizations are performed so far by the synthesis tools.

MPEG-4 AVC decoder	CAL	C actors	C scheduler
Number of files	43	83	1
Code Size (kSLOC)	5.8	44	0.9

Table 4: Code size and number of files automatically generated for MPEG-4 AVC decoder.

6. CONCLUSION AND OUTLOOK

This paper describes the essential components of the new ISO/IEC MPEG Reconfigurable Video Coding framework based on the streaming dataflow concept. The specification of the RVC MPEG tool library, that unifies and covers in modular form all MPEG video coding algorithms contained in the monolithic specifications of the different existing MPEG video coding standards, shows that dataflow programming is an appropriate way to build complex heterogeneous systems from high level system specifications. The MPEG RVC framework is supported by a simulator, software and hardware code synthesis that provide the low level implementation of the actors and associated network of actors for the different target implementation platforms (multi-core processors or FPGA). The RVC-CAL dataflow descriptions provided by the MPEG RVC standard result very synthetic and expressive if compared to equivalent specifications in the form of classical imperative languages. Therefore, the languages and tools used for MPEG-RVC are likely to be particularly efficient also for specifying other streaming and complex signal processing systems. Moreover, RVC-CAL libraries can also be developed in the form of proprietary implementations libraries to efficiently exploit architectural features of the target implementation platform in the synthesis process.

Extensions and improvements of the simulation and synthesis tools supporting the RVC framework are currently in development. They include the evolution of the software and hardware code generators to support SW and HW codesign, extended subsets of CAL language supported by the two code generators, the development of scheduling tools for mapping on multicore/codesign platforms and the evolution of the current Open DataFlow environment and its Open RVC-CAL Compiler infrastructure with more accurate and extended profiling and debugging tools. The LLVM backend is also a promising solution to create efficient, adaptive and portable video decoder implementations for a wide variety of platforms. The potential advantage of such approach is to join the active research activities on LLVM and MPEG RVC implementations, each contributing to achieve higher efficiency of RVC Decoder implementations. In the future, the LLVM integrated Virtual Machine will help to develop the process capable of dynamically and efficiently instantiate

[5]Open RVC-CAL Compiler: http://orcc.sf.net.

video coding tools. By combining the LLVM and the RVC concept, a portable and universal MPEG decoder engine that can dynamically configure a decoder implementation according to a MPEG RVC description could be created.

7. REFERENCES

[1] I. Amer., C. Lucarz, G. Roquier, M. Mattavelli, M. Raulet, J. Nezan, and O. Deforges. Reconfigurable video coding on multicore: an overview of its main objectives. *Signal Processing Magazine, IEEE*, 26(6):113–123, 2009.

[2] S. Bhattacharyya, J. Eker, J. Janneck, C. Lucarz, M. Mattavelli, and M. Raulet. Overview of the MPEG reconfigurable video coding framework. *Journal of Signal Processing Systems*, 2009.

[3] S. S. Bhattacharyya, G. Brebner, J. W. Janneck, J. Eker, C. von Platen, M. Mattavelli, and M. Raulet. OpenDF: a dataflow toolset for reconfigurable hardware and multicore systems. *SIGARCH Comput. Archit. News*, 36(5):29–35, 2008.

[4] D. Ding, L. Yu, C. Lucarz, and M. Mattavelli. Video decoder reconfigurations and AVS extensions in the new MPEG reconfigurable video coding framework. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 164–169, 2008.

[5] J. Eker and J. W. Janneck. CAL Language Report Specification of the CAL Actor Language. Technical Report UCB/ERL M03/48, EECS Department, University of California, Berkeley, 2003.

[6] J. Gorin, M. Raulet, Y.-L. Cheng, H.-Y. Lin, N. Siret, K. Sugimoto, and G. Lee. An RVC dataflow description of the AVC Constrained Baseline Profile decoder. In *IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding*, Cairo, Egypt, 2009.

[7] C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.

[8] IEEE. IEEE Std 1666 - 2005 IEEE Standard SystemC Language Reference Manual. *IEEE Std 1666-2005*, 2006.

[9] International Standard ISO/IEC 13818-1: 2000. *Information technology - Generic coding of moving pictures and associated audio information: Systems.*

[10] International Standard ISO/IEC 14494-1: 2004. *Information technology - Coding of audio-visual objects - Part 1: Systems*, 2004.

[11] International Standard ISO/IEC 23001-4:2009. *MPEG systems technologies – Part 4: Codec Configuration Representation*, 2009.

[12] International Standard ISO/IEC 23002-4:2009. *MPEG video technologies – Part 4: Video tool library*, 2009.

[13] International Standard ISO/IEC FDIS 23001-5:2008. *MPEG systems technologies - Part 5: Bitstream Syntax Description Language (BSDL).*

[14] E. S. Jang, J. Ohm, and M. Mattavelli. Whitepaper on Reconfigurable Video Coding (RVC). In *ISO/IEC JTC1/SC29/WG11 document N9586*, Antalya, Turkey, 2008.

[15] J. Janneck, I. Miller, D. Parlour, G. Roquier, M. Wipliez, and M. Raulet. Synthesizing hardware from dataflow programs. *Journal of Signal Processing Systems*, 2009.

[16] G. Kahn. The semantics of a simple language for parallel programming. In J. L. Rosenfeld, editor, *Information processing*, pages 471–475, Stockholm, Sweden, Aug 1974. North Holland, Amsterdam.

[17] G. Kahn and D. B. MacQueen. Coroutines and networks of parallel processes. In *IFIP Congress*, pages 993–998, 1977.

[18] C. Lattner and V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*, Palo Alto, California, Mar 2004.

[19] E. A. Lee. The problem with threads. *IEEE Computer Society*, 39(5):33–42, 2006.

[20] E. A. Lee and T. M. Parks. Dataflow Process Networks. *Proceedings of the IEEE*, 83(5):773–801, May 1995.

[21] C. Lucarz, I. Amer, and M. Mattavelli. Reconfigurable Video Coding: Concepts and Technologies. In *IEEE International Conference on Image Processing, Special Session on Reconfigurable Video Coding*, Cairo, Egypt, 2009.

[22] W. Plishker, N. Sane, M. Kiemb, K. Anand, and S. S. Bhattacharyya. Functional DIF for Rapid Prototyping. In *Proceedings of the 2008 The 19th IEEE/IFIP International Symposium on Rapid System Prototyping - Volume 00*, pages 17–23. IEEE Computer Society, 2008.

[23] M. Raulet, J. Piat, C. Lucarz, and M. Mattavelli. Validation of bitstream syntax and synthesis of parsers in the MPEG Reconfigurable Video Coding framework. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 293–298, 2008.

[24] G. Roquier, M. Wipliez, M. Raulet, J. W. Janneck, I. D. Miller, and D. B. Parlour. Automatic software synthesis of dataflow program: An MPEG-4 simple profile decoder case study. In *Signal Processing Systems, 2008. SiPS 2008. IEEE Workshop on*, pages 281–286, 2008.

[25] D. Rubel. The heart of eclipse. *Queue*, 4(8):36–44, 2006.

[26] M. Wipliez, G. Roquier, and J. Nezan. Software code generation for the RVC-CAL language. *Journal of Signal Processing Systems*, 2009.